

①

NAVAL POSTGRADUATE SCHOOL

Monterey, California

AD-A285 185



THESIS

DEVELOPING AN OBJECT-ORIENTED CURRICULUM

by

Curtis Howard Loehr

September, 1994

Thesis Advisor:

C. Thomas Wu

Approved for public release; distribution is unlimited.

94-31369



REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE Sep 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis	
4. TITLE AND SUBTITLE Developing an Object-Oriented Curriculum			5. FUNDING NUMBERS	
6. AUTHOR(S) Curtis H. Loehr				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSORING/MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) Traditional introductory computer science curricula do not address the emerging paradigm of object-oriented programming. The purpose of this research is to determine when object-orientation should be introduced into the computer science curriculum and what is the proper instructional approach to present this material. This thesis looks at the concepts incorporated by the object-oriented paradigm, explores the developmental psychology applicable to understanding new environments and proposes an introductory object-oriented curriculum that incorporates the fundamentals of learning, computer science and object-oriented programming. The object-oriented curriculum proposed provides a top-down approach to the conceptual foundations of computer science with a bottom-up approach to object-oriented programming. The combination of approaches provides the necessary breadth of coverage in algorithms, data structures, programming analysis and object-oriented modeling with an initial in-depth look at the mechanics of programming.				
14. SUBJECT TERMS object-oriented programming instruction, computer science curriculum, teaching introductory computer science			15. NUMBER OF PAGES 86	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL	

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18

Approved for public release; distribution is unlimited.

Developing an Object-Oriented Curriculum

by

Curtis H. Loehr
Lieutenant, United States Navy
B.S., Michigan State University, 1986

Submitted in partial fulfillment
of the requirements for the degree of

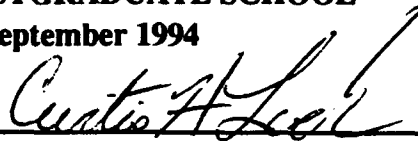
MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

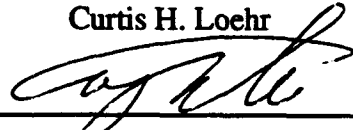
September 1994

Author:



Curtis H. Loehr

Approved by:



C. Thomas Wu, Thesis Advisor



Lt Col David A. Gaitros, USAF, Second Reader



Ted G. Lewis, Chairman
Department of Computer Science

ABSTRACT

Traditional introductory computer science curricula do not address the emerging paradigm of object-oriented programming. The purpose of this research is to determine when object-orientation should be introduced into the computer science curriculum and what is the proper instructional approach to present this material.

This thesis looks at the concepts incorporated by the object-oriented paradigm, explores the developmental psychology applicable to understanding new environments and proposes an introductory object-oriented curriculum that incorporates the fundamentals of learning, computer science and object-oriented programming.

The object-oriented curriculum proposed provides a top-down approach to the conceptual foundations of computer science with a bottom-up approach to object-oriented programming. This combination of approaches provides the necessary breadth of coverage in algorithms, data structures, programming analysis and object-oriented modeling with an initial in-depth look at the mechanics of programming.

Accession For	
NTIS	CRA&I <input checked="checked" type="checkbox"/>
DTIC	TAB <input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

TABLE OF CONTENTS

I. INTRODUCTION	1
A. PROBLEMS WITH STRUCTURED PROGRAMMING	2
B. COMPLEXITY MANAGEMENT	3
1. Abstraction	4
2. Encapsulation	4
3. Reusability	5
4. Extendibility	6
5. Maintainability	6
C. THESIS MOTIVATION	7
D. THESIS ORGANIZATION	8
II. OBJECT-ORIENTED CONCEPTS	9
A. OBJECTS	9
1. Object Definition	10
2. Identity	12
3. Persistence	12
4. Distinct Concepts	13
a. Objects and Programs	13
b. Objects and Data	13
B. CLASSES	13
1. Class Definition	14
2. Classes as ADTs	16
3. Classes, Encapsulation and Abstraction	17
C. INHERITANCE	18
1. Inheritance Definition	18
2. Specialization	19
3. Multiple Inheritance	20
D. AGGREGATION	21
E. POLYMORPHISM	22
1. Abstract Aspects	22
2. Names	23
3. Additional	23

F. CONCLUSIONS	23
III.SURVEY OF OBJECT-ORIENTED LANGUAGES	25
A. SMALLTALK	25
B. C++	28
C. ADA 9X	30
D. LANGUAGE CONCLUSIONS	33
IV. PIAGET'S DEVELOPMENTAL PSYCHOLOGY	35
A. SENSORIMOTOR INTELLIGENCE	35
B. REPRESENTATIVE INTELLIGENCE AND CONCRETE OPERATIONS	36
1. Pre-operational Phase	36
2. Concrete Operations	37
C. FORMAL OPERATIONS	38
D. MAPPING PIAGET'S DEVELOPMENTAL PSYCHOLOGY TO TEACHING OOP	39
1. Mapping	41
V.APPROACHES TO TEACHING OOP	45
A. PRINCIPLES OF INSTRUCTIONAL DESIGN	45
1. Aid the Individual	46
2. Short and Long Term Outlook	46
3. Systematic Instruction	46
4. Systems Approach	47
5. Developmental Psychology	47
B. INSTRUCTIONAL SYSTEM DEVELOPMENT	47
1. System Level	48
2. Course Level	48
3. Lesson Level	49
C. EVOLUTION OF COMPUTER SCIENCE CURRICULUM	50
1. Curriculum 68	50
2. Curriculum 78	50
3. The Liberal Arts Model Curriculum	51
4. Denning Report	52
5. Curriculum 91	53
D. APPROACHES TO CURRICULUM DEVELOPMENT	53
E. APPROACHES TO TEACHING OOP	55

1. Top-down/Bottom-up	56
2. Pure/Hybrid Languages	57
F. DESIGN CONCLUSIONS	57
VI. PROPOSED CURRICULUM	59
A. INTRODUCTION	59
B. SYSTEM LEVEL REFORM	60
1. Standard Curricula Model	60
2. Computer Science Curricula	61
C. COURSE LEVEL REFORM	62
D. INTEGRATING APPROACHES	63
1. Bottom-up and Top-down	63
2. Inclusion of CS and Non CS Students in CS1	64
E. PROPOSED CURRICULUM	64
1. A Model For CS1	65
a. Introduction To Computing	66
b. Introduction To Programming	66
1) Stage 1. Non-OOP	67
2) Stage 2. Semi-OOP	67
3) Stage 3. Full OOP	68
c. Intermediate Courses	68
F. CONCLUSIONS	69
VII. CONCLUSIONS	71
A. OO CONCEPTS AND LANGUAGES	71
B. THE PSYCHOLOGY OF LEARNING	72
C. OO INTEGRATION INTO THE CS1 CURRICULUM	73
LIST OF REFERENCES	75
INITIAL DISTRIBUTION LIST	79

I. INTRODUCTION

Object-oriented (OO) concepts and methodologies are now in the mainstream of software development. These concepts represent a complete approach for planning, designing and implementing solutions for the expanding range of complex problems. Structured languages, while still maintaining a vital role in software systems, are moving toward OO concepts to remain competitive. Although science and industry are eagerly accepting the OO paradigm, universities and graduate schools have not formed a comprehensive curriculum that fundamentally teaches object-oriented programming (OOP).

This thesis will look at why the OO paradigm is the programming methodology of the future, discuss and contrast some principle OC languages, and develop the logical basis for concept instruction. The main focus of this thesis is to propose an OO curriculum that incorporates these principles.

The problems concerning structured programming and why OOP is the method of the future are briefly reviewed in this chapter. Some of the beneficial properties of the OO approach to software development are not unique to the OO philosophy or to all object-oriented programming languages (OOPL). Looking at the problems encountered in structured programming and the concepts and facilities the OO philosophy brings to the software lifecycle provides a appropriate background for the importance of developing an OO syllabus.

A PROBLEMS WITH STRUCTURED PROGRAMMING

In conventional structured programming, data structures and behavior are only loosely connected. In typical structural methodologies the main emphasis is placed on specifying and decomposing system functionality. This system is more direct and leads to goal implementation, unfortunately if the system changes, modifications are not easily made. The software lifecycle stages are normally requirements, design, implementation, testing, and maintenance. Most of the cost of software is spent on maintenance, between 50% and 75% (Lehman, 1980).

Maintenance can be divided into three sub-activities:

- Corrective Maintenance - performed in response to the assessment of failures;
- Adaptive Maintenance - performed in anticipation of change within the data or processing environment;
- Perfective Maintenance - performed to eliminate inefficiencies and enhance performance or improve maintainability. (Lientz, Swanson, Tompkins, 1978)

Software enhancement is the largest portion of system maintenance. Sixty percent of all maintenance money is spent on perfective maintenance (Fairley, 1985). That is the maintenance required to improve functioning software after it has been delivered. If you can improve the maintenance phase of the software lifecycle, especially the ability to enhance software, the overall cost of software will be greatly reduced. This is obviously of great interest to business, government and academia.

Object-oriented technology specifies what an object is, instead of just how it is used. The way an object is used depends on the details of the application. These details frequently change during and after development. As requirements evolve, the features

supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run (Booch, 1986). Object-oriented development places a greater emphasis on data structure and a lesser emphasis on procedure structure than traditional functional methodologies. The object-oriented methodology focuses on identifying objects from the application domain, then fitting procedures around them. Although this is more indirect, object-oriented software holds up better as requirements evolve because it is based on the underlying framework of the application domain itself rather than the ad-hoc functional requirements of a single problem.

B. COMPLEXITY MANAGEMENT

Software development increasingly occurs in an industrial setting typified by product complexity, system longevity, and incessant product evolution (Jacobson, 1991). OO techniques have been employed for developing complex software products such as compilers, databases, computer aided design (CAD) systems, simulations, meta models, operating systems, spreadsheets, signal processors, and control systems (Rumbaugh, 1991). Development of such complex systems requires architectures, methods, and processes that divide system development into smaller parts and that can handle change efficiently (Jacobson, 1991). In the next subsections I will explain some of the desirable features the OO methodology contributes to complexity management.

1. Abstraction

Abstraction is used to simplify the design of a complex system by reducing the number of details that must be considered at the same time (Berzins, 1991). Abstraction allows us the ability to simplify complex objects. By simplifying the object, knowledge is expressed as generalized essential information which can then be better understood.

The level of detail necessary to formulate an abstraction varies with the requirements for the problem (Booch, 1987). OO analysis, design, and programming use abstraction to focus attention on the behaviors and attributes of objects rather than on the implementation details. Using this method of thinking, problem entities can be pursued with successive levels of refinement. Each refinement is an abstraction of a particular level of detail. This allows designs to be conceived as multilevel structures of abstractions.

2. Encapsulation

Information hiding emphasizes the need to separate function from implementation. Apart from continuity, it is also related to the requirements of de-composability, composability and understandability: to separately develop the modules of a system, to combine various existing modules, it is indispensable to know what each of them may and may not expect from the others. (Meyer, 1988)

Encapsulation is a technique for minimizing interdependencies among separately written modules (Snyder, 1986). A external interface is used to allow interaction between data structures and function implementation. In this way the knowledge about data structures is kept private. In the context of software development, encapsulation promotes the independent construction of cooperating modules and isolates the effects of

implementation modifications to the affected areas only. Implementation details can be modified without impinging on the users of the interface, so long as external interfaces are stable. This feature allows software maintenance to become localized and avoids the perilous search for links between interrelated program modules. The implementation of this feature contributes to savings of time, money, and human resources. As such, encapsulation is a critical measure of any OOPL.

3. Reusability

Reuse may be defined as the effective ability to incorporate objects created for one software system into a different software system. The essence of reuse is the ability to take all or part of a product and completely and correctly embed it within a new product that may be constituted and structured quite differently. (Wasserman, 1991)

Reusability is a language property that allows previously developed software to be incorporated into new software. The benefits of reuse are mainly: (1) development effort is reduced; (2) reused code has already been tested and verified. The principal OO mechanisms for achieving reuse are inheritance, polymorphism, and dynamic binding. Much of the value of programming in the OO environment arises from the capability to use previously developed code stored in software libraries. Developers may also be familiar with a problem's requirements and important abstractions; consequently, opportunities for reusing not only software, but entire designs and requirements also exist. (Booch, 1991)

4. Extendibility

Extendibility is the ease with which software products may be adapted to changes in requirements (Meyer, 1988). Extendibility is a concept allied to reusability. It encompasses those properties which enable new code to be developed as extensions to previously written code. Extendibility assumes greater importance as problem understanding improves. This results in possible new requirements. As program scale grows, extendibility is best achieved through design simplicity and modular decentralization (Meyer, 1988). In the OO environment, extendibility is realized through the application of inheritance techniques to class definitions in class hierarchies.

5. Maintainability

A designer endeavors to organize a design so that it is resilient to change; a packaging that will remain stable over time is sought. The answer is to separate those parts of the system that are intrinsically volatile from those parts that are likely to be stable. (Coad and Yourdon, 1991)

Maintainability refers to the efficiency with which modification can be introduced over the software lifecycle. It is an economic issue which concerns the degree to which linkages in program elements magnify the effects of modifications. Economically, maintainability reflects the cost required to correct, modify or extend code. Software that exhibits strong abstraction, encapsulation, reusability and extendibility generally has favorable maintainability qualities.

C. THESIS MOTIVATION

OO technology as been accepted in the mainstream professional community. As industry continues to lead the way in OO use and development, academia presents students with out dated programming environments. For several years, calls have been made to incorporate this methodology into the undergraduate curriculum (Temte, 1991). The claim has been made that OO technology will become the dominant software development methododology, replacing the traditional decomposition model (Lutz, 1990). Undergraduate CS programs are contemplating introducing or extending the OO paradigm into the traditional CS curriculum. There has yet to be a widespread concerted effort in the educational community to amend curricula in order to accommodate object-orientedness (Temte, 1990). Academic environments given limited time and resources are questioning instructional policy that provide OO concepts only as advanced courses to the primary structured curriculum. A CS curriculum is required that will integrate the OO paradigm at the initial stages of instruction.

The purpose of this thesis is to review the concepts of OOP and explain the benefits of this methodology, to compare these concepts and how they are implemented in three popular OO languages, to look at the developmental stages of learning to better understand the proper presentation of concepts and finally to propose a curriculum that will introduce the OO paradigm in the initial stages of CS education.

D. THESIS ORGANIZATION

Chapter II reviews the OO literature to highlight definitions of the fundamental concepts. These definitions are also the critical items initially to be covered in any substantive OO curriculum. Chapter III draws upon the OO literature to contrast three OO languages and how they incorporate the essential definitions. Chapter IV looks at the Developmental Psychology work authored by Jean Piaget. His work outlines the logical progression of learning which leads to insights when preparing an OO teaching syllabus. Chapter V examines the basic principles of instruction, various approaches to teaching programming, the evolutions of the computer science curriculum and some alternatives approaches within the OO methodology. Chapter VI proposes the curriculum and introduces the teaching syllabus. Finally, Chapter VII offers conclusions and suggestions for implementation and further research.

II. OBJECT-ORIENTED CONCEPTS

The object-oriented concept represents a complete philosophy for planning, designing and implementing solutions to complex problems. Development of such complex systems requires architecture's, methods, and processes that divide system development into smaller parts and that can handle change efficiently. (Jacobson, 1991)

The following concepts are the desirable features that OO methodologies attempt to bring to software development for the managing of complex systems. Although there currently are no OO standards, there is consensus as to the primary concepts which formulate the OO paradigm. This chapter will review some of the benefits of the OO methodology, define some of the fundamental concepts and review the beneficial properties of the OO approach to software development.

A. OBJECTS

Objects have a unusual dual status within the OO system. In one instance they are introduced as physical entities in the problem domain. Alternatively, they are presented as the primary programming constructs which closely parallel the constructs in the problem domain. In terms of practical consequences the difference between the two is minimal, yet the distinction should be noted as the latter emphasizes that objects are constrained not only by possibilities from the real-world but also by the capabilities of the programming language.

1. Object Definition

An object has state, behavior, and identity; the structure and behavior of similar objects are defined by their common characteristics; the terms instance and object are interchangeable (Booch, 1991). Objects are entities that combine the properties of procedures and data since they perform computations and save local state (Stifik and Bobrow, 1986).

Objects have a structure which preserves the state of an object. An object may change states over the course of its existence, so, objects can have a history. Objects also exhibit a observable behavior. Objects communicate with each other by passing messages to request desired behavior. Objects have an identity that distinguishes each object from all others. (Loomis, 1991)

In order to define an object you must first find sets or classes of objects with a common structure and behavior. They are the objects or concepts from the real world enterprise which is being modeled. This is a form of data abstraction where something is represented if it contains a set of similar objects or concepts with meaningful properties and operations that are required to be maintained by the system. Therefore a class is the abstraction of shared characteristics. This is similar to the approach used in the entity relationship modeling concept except in object oriented modeling the design is not constrained by implementation or normalization considerations nor does it strictly pertain to database relationships. The attributes of object classes need not be non-decomposable

or single valued. Objects relevant to the organization being modeled are organized into different categories. (Bertion, 1991)

Families of objects have common traits: Inheritance is used to model objects in order to reduce the need to duplicate shared properties and operations. The existence of an inheritance hierarchy is indicated by the presence of properties or operations which only apply to certain instances of an object class.

Part/whole relationships: Objects that have an "is-part-of" relationship with another object. This is a complex object, an object which has a complex structure consisting of other sub-objects. For example the object Engine is part of the complex object Vehicle.

Groups or Clusters of closely inter-related classes that together describe a part of the system. Clustering is an aid to both conceptual modeling of large systems and also to physical implementation. Objects that tend to be accessed together are placed near each other in physical storage.

General purpose classes that are used by many applications that are called base classes. Base classes may be available from class libraries which provide ready to use abstractions for commonly used data structures. (Bertion, 1991)

The behavior of an object is the set of actions an object can undertake. During a program an object sends a message to another object requesting a service offered by the receiving object. The receiving object determines how best to comply with the request, selecting among a set of methods which satisfy the request. Much of the versatility and

confusion in OOP comes from the mechanisms that determine which object receives a request and which method is selected.

2. Identity

Objects as programming constructs achieve a high level of abstraction and closely parallel their real-world counterparts. A requirement for the computer environment is to have "...the ability to distinguish objects from one another regardless of their content, location or addressability, and to be able to share objects (Khoshafian and Copeland, 1986). Object identity enables us to realize this goal. An argument can be made that an OO language must maintain identity despite changes in an object's state, address, or user-defined name, and throughout its lifetime (Khoshafian and Copeland, 1986). This is accomplished in an OO language by maintaining an identifier built into the object that will not change. The failure to recognize the difference between the name of an object and the object itself is the source of many kinds of errors in object-oriented programming (Booch, 1991). These errors include assignment operations which orphan objects, using aliases through assignment (structural sharing), and inappropriate semantics for equality operators (Booch, 1991).

3. Persistence

An object's existence does not necessarily depend on the program from which it was created. The fact that objects have an identity and can have a history implies that it can exist beyond the lifetime of the program(s) in which the object may have been created or used. This quality is termed persistence. (Loomis, 1991)

4. Distinct Concepts

a. Objects and Programs

Construction of a program using the OO paradigm produces a different perspective than structural methods. This traditional approach to programming consists of procedural modules which act on data. Programming from this perspective leads to a top-down, functional decomposition of programs. The OO methodology consists of objects acting in cooperation, but independently. This approach can achieve various levels of integration, producing system behavior at high levels. This approach allows complex systems to be modeled as interacting objects.

b. Objects and Data

Objects are not just data structures. Objects are entities which have both structure and behavior. The implementation of an object's structure should be encapsulated, although this is not always possible in OO languages, nor is it available in data driven programs in which data structures are globally accessed and modified.

B. CLASSES

The evolution of software engineering has brought the modularization of software components. Berzins and Luqi define a module as a "...conceptual unit in a software system that corresponds to a clearly identifiable region of the program text." (Berzins and Luqi, 1991)

From this view, modularization is a part of software construction which produces "...software systems made of autonomous elements connected by a coherent, simple structure." (Meyer, 1988) Modularization promotes conceptual localization of code.

This allows the realization of desirable properties including data abstraction, encapsulation, reusability, extendibility, reliability, and maintainability. Although objects are declared as instance of classes, it is classes that are recognized as the key modules in most OO languages. Classes are the key design modules and objects, as instance of classes, are the key program modules.

1. Class Definition

A class is a template from which objects may be created by 'create ' or 'new' operations. Objects of the same class have common operations and therefore uniform behavior. (Wegner, 1987) Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the 'essence' of an object, as it were (Booch, 1991).

Object oriented systems use two different but related mechanisms for representing objects and sharing behavior, based upon either classes or prototype. There are two categories of objects, classes and instances. A class acts as a template for a set of instances, describing their structure and behavior. Classes can contain values, methods, and programs. Instances do not have to have values for all the properties described by their class, but they cannot have any properties which are not declared in their class. Additionally a characteristic of a class is that all the instances of the class are stored with the class descriptor. This means that it is straightforward to iterate over all instances of a class as they are at least conceptually stored together. A class can be seen as an object factory, which indicates how an object is made, plus an object warehouse where the

objects are stored (Bancilhon, 1988). An object is an instance of a class and therefore a basic run time entity. The instances of a class share attribute definitions, not the values.

An alternative to the class based object oriented system is to use prototypes in which there is only one category of object. The distinction between class and object is gone. Prototypes involve generating a new object starting from another existing object by modifying its attributes and/or its behavior. A prototype is an individual object containing its own description. A prototype can also be used as a model for creating other objects. This can be useful when objects evolve quickly and have more differences than similarities. Prototyping is also useful when there are a few instances for each class. The proliferation of many classes each with a few instances is avoided.

A class construct supports encapsulation through the separation of the class interface and class implementation. Such separation permits the class interface to be mapped into several different implementations and at the same time makes sure that the operators in the external interface represent the possible behaviors of that object. One of the responsibilities of such operators is to provide for controlled access to the attributes of the object which would be hidden from the users of the class. (Chorafas and Steinmann, 1993)

An interface to a class consists of those variables and methods which are visible to other objects and to subclasses. The interface available to other objects is called the external view and the interface available to subclasses is called the internal view (Micallef, 1988).

The ability to limit the various interface visibility's is not present in every OOPL. Languages like C++ provide a mechanism for enforcing private and public distinctions. This private area represents knowledge that is not available to other objects. It is not part of the external or the internal interface. Public areas (variables and methods) are integrated into the external and internal interfaces. To enforce encapsulation variables are kept private, accessible only through public methods. Internal interfaces can apply an additional level of control by declaring variables and method protected. This makes them invisible to other objects, but not to their subclasses.

2. Classes as ADTs

Data abstraction is defined as "...the principle of defining a data type in terms of the operations that apply to objects of the type, with the constraint that the values of such objects can be modified and observed only by the use of the operations." (Coad and Yourdon, 1991) When describing data structures it is desirable to have complete, precise, unambiguous descriptions that are not based on the physical representation of the underlying structure (Meyer, 1988).

In OO languages like C++ and Eiffel, classes are equivalent to ADTs. Classes, which are the modular units of interaction, take a specific purpose: the description of data types. The interaction between modules (classes) are managed through the type interfaces. Nevertheless, it is important to understand that the principle function of classes is to serve as templates for object instantiation and not as predicate descriptors. (Wegner, 1988)

3. Classes, Encapsulation and Abstraction

Abstraction and encapsulation are complementary concepts. Abstraction looks at the outside view of an object. An abstraction denotes the essential characteristics of an object that distinguish it from other kinds of objects and thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer. (Booch, 1991)

The abstraction of an object should precede the decisions about its implementation. Once the implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients. No section of a large complex system should depend on the internal details of any other section. Abstraction is used to simplify the design of a complex system by reducing the number of details that must be considered at the same time. (Berzins, 1991)

As modular software components, classes implement details of structure and behavior. Modularization permits the design of interfaces which encapsulate these implementation details. This achieves the many benefits attributed to encapsulation. Encapsulation represents a property, not a responsibility of classes. It is the programmers responsibility to specifically design interfaces which segregate implementation from specification. The philosophy of OO languages must support encapsulation by restricting access/manipulation of data structures of the designed interface. Not every language that supports classes enforces encapsulation. Instance variables in Simula are directly accessible (Micallef, 1988). This increase the linkages among program modules, reduces reliability of code, and increases the difficulty of maintenance.

C. INHERITANCE

Inheritance uniquely distinguishes OO languages from other programming languages. In the family of OO languages the inheritance mechanisms vary widely. Inheritance is a large concept which serves multiple ends and should be looked at from different perspectives to fully understand its concepts.

1. Inheritance Definition

Inheritance enables the easy creation of objects that are almost like other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place. (Stefik and Bobrow, 1986)

We adopt the view of Cook who defines inheritance as a composition mechanism that internalizes inherited attributes by late (execution time) binding of self-reference to the inheriting object (Wegner and Zdonik, 1988).

Inheritance is primarily a resource sharing mechanism, greatly extending reusability. Defined objects are organized in a hierarchy, allowing operations implemented by a parent type to be inherited and reused by a child type. This promotes uniformity among types and affords the advantage of being able to represent knowledge at the highest level of abstraction. It also helps to maintain consistency of the knowledge base when adding new objects or concepts.

Groups of classes can manifest commonalties which result in hierarchical relationships among class definitions. The concept of inheritance is the second reusability mechanism. Inheritance lets a class be defined starting from the definition of another class,

called the superclass. A subclass inherits the superclass attributes, methods, and messages. A subclass can have specific attributes, methods and messages that are not inherited. A subclass can override the definition of the superclass attributes and methods. The inheritance mechanism lets a class specialize another class by additions and substitutions. Inheritance represents an important form of abstraction since the differences of many class descriptions are abstracted away and the similarities factored out as a more general superclass.

OO languages can implement single inheritance in which a subclass is only allowed to inherit from a single superclass or multiple inheritance in which a subclass inherits from one or more superclasses. Multiple inheritance greatly increase the opportunities for code reuse, but it also introduces several complications. Solutions for these complication vary from language to language.

2. Specialization

There are different strategies used to design class hierarchies. The possibilities include, specialization, type and like hierarchies. Classes may show no abstract commonalties other than code sharing or interface sharing.

Specialization is described as the primary principle for hierarchy design although a consistent formula for building such hierarchy has not been generally accepted. Specialization hierarchies are also called 'is_a' hierarchies. The 'is_a' hierarchies consist of "...superclasses representing generalized abstractions, and subclasses representing

specializations in which fields and method from the superclass are added, modified, or even hidden." (Booch, 1991)

Exactly what qualifies as specialized behavior? What are the mechanisms which implement inheritance and the abstractions which relate classes in a specialization hierarchy? A standardized notion of specialization, based upon some philosophical foundation is required to introduce continuity to hierarchy construction and to facilitate the construction of compatible hierarchies. This is central to forming OO libraries and code reuse.

3. Multiple Inheritance

Multiple inheritance allows a class to have more than one superclass and to inherit features from all parents. A subclass may inherit from several superclasses. The 'is-a' relationship should guide the construction of multiple inheritance hierarchies (directed acyclic lattices), noting, however, that the resulting subclass should be viewed as a specialized "...combination or collection of several different components." (Budd, 1991)

Multiple inheritance introduces new problems. Name conflicts and inheritance from common ancestors are the most prominent. Name conflict resolution strategies have been proposed which provide a useful framework for analyzing such conflicts. Knudsen distinguishes horizontal from vertical name collision. Conflicts can be characterized in three ways: (1) the same phenomena are defined; (2) casually related phenomena are defined; and, (3) unique phenomena are defined in which no collisions are permissible (Knudsen, 1988). The first method is handled by polymorphic techniques, the second by

resolution operators and the third will give compile-time errors. Inheritance from a common ancestor involves inheritance of attributes from superclasses whose inheritance paths converge at a common ancestor.

D. AGGREGATION

Aggregation is the "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly (Rumbaugh, et al, 1991). An aggregation relationship relates an assembly class to one component class. Complex objects can be conceived as consisting of aggregates of other objects. The object is 'part-of' another object. Composite objects are a group of interconnected objects that are instantiated together, a recursive extension of the notion of object (Stefik and Bobrow, 1986). This presents several ideas from the concept of composite objects.

First, composition is another mechanism for reusability. Redefinition is not necessary if the class template for a group of objects is included from a previously defined template. These composition relationships can be implemented through two mechanisms: (1) declaration of class instance variables as user defined types; and (2) declaration of formal parameters for class methods as user defined types (as a parameter to the class interface). (Booch, 1991)

Second, composition should not be confused with single or multiple inheritance. The subclass inherits from a superclass only once while aggregation allows more than one instance of particular object type." (Halbert and O'Brien, 1987)

Third, the notion of composition as a recursive definition highlights the fact that members of a composite object may themselves additionally be composite objects. Any level of complexity is possible.

E. POLYMORPHISM

Polymorphism is a concept from which it is difficult to obtain a clear understanding. To achieve reusability inheritance, specialization, message passing and polymorphism all interact. This tends to complicate the isolation of the content and effects of polymorphism. The definitions of polymorphism often overlap other concepts such as overloading.

1. Abstract Aspects

In programming languages, "a polymorphic object is an entity, such as a variable or function argument, that is permitted to hold values of differing types during the course of execution." (Budd, 1991) Most OO programming languages provide an efficient message passing construct that enables receivers of messages to change (Ingalls, 1986). In a strongly typed environment such as C++ , the changing among types or message receivers is constrained by inheritance (Meyer, 1988).

Polymorphism is a group of mechanisms that permit programming constructs (method names, method arguments, and objects) to shift definitions in the course of program execution. This mechanism is different for each programming language. Some languages distinguish the static, declared class of an object from the dynamic class of its value (Meyer, 1988). Polymorphism can be managed by manipulation of references and pointers or by binding values to objects at run-time.

2. Names

Polymorphic names occur when the same message is sent to different objects. This is referred to as overloading of function names. Several classes may have a method with the same name. Additionally, methods with the same names can have different argument cardinality or different argument types. The methods are all grouped within a single class. An example is the constructor function in C++ classes.

3. Additional

These polymorphic forms are the basic cases found in most OO languages. Additionally polymorphic forms include overriding, virtual, deferred, and parametric techniques.

F. CONCLUSIONS

The OO paradigm has changed since the introduction of the first OO language CEMBALO in 1968 (Meyer, 1988). Inheritance uniquely distinguishes OO languages from other programming languages. Languages which include objects and classes, but not inheritance are called object-based languages (Ada 83). Conceptual standardization is what currently restricts the OO paradigm from becoming the methodology of choice.

III. SURVEY OF OBJECT-ORIENTED LANGUAGES

The introduction of object-oriented languages and variants has continued to grow into the 1990's. The once limiting obstacle of memory and MIPS intensive OOPL's has been removed by the growth in capability of computer hardware. The growth of OOPL rose in three separate, yet interconnected strains. Those were LISP-based, Smalltalk-based and C-based. (Saunders, 1989)

This chapter looks at three object-oriented programming languages. Smalltalk is a widely used "pure" OOPL, C++ is the commercially successful OO addition to the family of C languages, and ADA-9x is the military supported entrant into the OO community.

All these languages possess the following built in characteristics:

1. Object creation facility
2. Message passing capability
3. Class capability
4. Inheritance feature (Saunders, 1989)

A. SMALLTALK

Smalltalk features pure OOP which provides a interesting dimension in which to organize the elements of a software systems. This allows the programmer to create highly reusable software, generic code and the opportunity to use a prototyping style of software development.

There are some specific features that make Smalltalk a popular OOPL. Smalltalk has a seamless programming environment. It encourages programming by modification though code re-use. Using Smalltalk, it is possible to introduce the notions of

encapsulation and procedural abstraction early in an introductory course. Students develop code using interactive browsers and a source code debugger. An extensive class library is available for access to source code and to classic approaches with common syntax.

In the context of an introductory computer science course, a solid foundation is necessary in the fundamentals of using constructs such as variables, data types, iteration and conditional statements. The skills of documenting, debugging and testing are also required in an initial programming course. Smalltalk provides the ability to introduce these concepts in conjunction with the notions of encapsulation and procedural abstraction. The methodology for using Smalltalk consists of:

1. Identifying the objects appearing in the problem and its solution.
2. Classifying the objects according to their similarities and differences.
3. Designing messages which make up the language of interaction among the objects.
4. Implementing the methods (algorithms) that carryout the interaction among objects (Digitalk, 1991).

Smalltalk is relatively basic in its simple syntax and semantics. The concepts of objects, class, message and method form the basis of Smalltalk programming. All Smalltalk objects are abstract data types. Every object is an instance of a class. The class defines the structure and behavior of all its instances. The class of an object is determined by sending it the message class. Classes describe the data structure (objects), algorithms (methods) and interfaces (messages). Every object is an instance of some class. Objects that are instances of a certain class are similar, have the same instance variables, and respond to the same messages. The classes in Smalltalk form a hierarchy beginning with

the root class object. A class object provides common behavior for all objects. Each subclass builds on its superclass by adding its own methods and instance variables.

All Smalltalk variables are containers for objects. They contain a single object pointer. Messages in Smalltalk are equivalent to function calls. Objects and messages are safe. Objects have a state while messages are used to change that state. Methods are the algorithms that determine an object's behavior and performance. They are like function definitions. When a message is sent to an object, a method is evaluated and the result returned is an object. Class methods implement messages sent to the class. They respond to messages sent to class objects. The receiver of a class message is always sent to instances of the class. The receiver of an instance method is always an object that is an instance of the class. A message is sent to a class object to allocate a new object. It creates a new instance of a class.

Inheritance is provided by supplying the name of the supertype to an object. All attributes of a superclass are available to all descendants. Only those features that are unique to the subclass are specified. Smalltalk does not support multiple inheritance.

All Smalltalk objects are dynamic and allocated from a heap. Deallocation is performed by a built in garbage collector. Tools are available in Smalltalk for browsing, editing, compilation, debugging, system integration and testing. The browser also allows exploration of source code. Messages in Smalltalk are bound at run-time. In Smalltalk all operations are public, which is not conducive for large multi-person projects.

B. C++

C++ is the C language extended to support OOP. Its OO features have been well integrated into the base language and very little in the way of new syntax was added to provide the necessary support. The features that modern software engineering considers indispensable are present in C++.

C was chosen as the base language for C++ because it is versatile, relatively low-level and was running on most machines around the world. The decision to maintain compatibility with the C language was done to support the millions of lines of C code that may benefit from improvements in C++. There is an extensive set of library functions and utility software code written in C that is useful to C++. (Stroustrup, 1993)

There are many small changes in C++ from the ANSI version of C. The only major change is the addition of classes. In C++, classes are the storage regions in memory that allow the building of objects. The most significant changes noticeable in C++ programming as contrasted with C are:

1. A notion of distributed rather than centralized control
2. A more readily reasoned decomposition of a problem into modules.
3. The common use of dynamically instantiated objects.
4. The hiding of almost all global variables. (Reid, 1991)

The more flexible structure in the design of C++ supports the increase in the scale of programs written since C was first introduced. Good style and structure can be avoided in a small program and still produce sufficient code. Failure to maintain proper structure as the size of a program grows will guarantee a continuing progression of errors. The base language of C is designed so there is a very close correspondence between its types,

operators, and statements and the objects computers deal with directly, numbers, characters, and addresses (Stroustrup, 1993). The difference between C and C++ is primarily in the degree of emphasis on types and structure. C++ is much more expressive than C.

There are three features in C++ that are critical to providing OO design. The three features of C++ are constructors, which allow you to control how objects are created; templates, which let you create classes that have the same code but for different types; and friends, which relax the C++ access rules. These three topics are important C++ capabilities that have a major impact on all C++ OO software design.

C++ uses a special member function called a constructor to initialize data members. The constructor ensures that necessary initialization is performed when an object is created. Constructors can have arguments so objects can be constructed with a specified initial value. The point of construction is to ensure the necessary initialization chores are performed when an object is created.

A template allows the specification of classes or functions without filling in all the type information. At a later time the specific type of class or function can be created. Templates give you the ability to create generic solutions but with the advantage of having strongly typed interfaces.

The C++ public, protected and private access restrictions prevent unwanted access. This allows a class to enforce its own restrictions and rely on its own conventions. Classes can provide access to non-public members by using public access functions. This

gives a class control over how its non-public members are used. The C++ designers believed there are situations in which there is a legitimate need to access another class's non-public members. The C++ concept of friendship bypasses the access system providing direct access to the specified friends.

C. ADA 9X

Ada 9x , like C++ , is a modern general purpose language. It has roughly similar power aimed broadly at the area of systems programming. Ada 9x is also intended for embedded and real time systems, and has many features to support concurrent and distributed programming. Ada 9x has kept the best features of its predecessor Ada 83. Ada 9x and C++ have features that modern software engineering practice considers indispensable; modularity, information hiding, structuring tools for large programs, *inheritance and support of OO design methods.*

Both Ada 9x and C++ are superior to their competitors (C, Modula-2, Pascal, Eiffel) in terms of expressive power , maturity, and software base. Ada is also superior in terms of safety and reliability. Ada 9x has additional advantages over C++ in terms of software costs when these costs are examined over the lifetime of the software system. (Schonberg, 1992)

The comparison of programming languages is a subjective affair. Judgments are influenced by personal stylistic choices, by familiarity and by the first language effect. When comparing language issues, the arguments need to be pragmatic. Issues of

reliability, ease of use, modifiability of resulting code, the training of programmers, and the availability of tools should dominate the discussion.

OOP simplifies the design and construction of software systems through the reuse abilities of inheritance, specialization by extensions and dynamic dispatching (Anderson, 1992). The gain is in the amount of code that does not have to be rewritten. Ada 9x implements OOP by an extension of the notion of derived type. Objects are the same as in Ada 83, they are entities that can have values of a certain type. Ada 9x supports multiple inheritance via multiple "with/use" clauses, multiple inheritance of implementation via private extensions and record composition, multiple inheritance mix-ins via the "use" generics, formal packages and access discriminants (Anderson, 1992). The mechanisms in Ada 9x are designed to eliminate distributed overhead, so that there is no added expense for the general user because of the presence of the mechanisms supporting multiple inheritance (Anderson, 1992).

The concept of OOP brings with it the insight that types should be enriched by extending what they inherit, rather than being simply copies of their ancestors. Ada 9x implements OOP by a straightforward extension of the notion of derived type (AMSR, 92).

Objects are the same as in Ada 83, they can have values of a certain type. The conventional concept of class in Ada 9x is the type extension. The derived type inherits the primitive operations of its parent type. There is no special syntax to designate

objects. Ada 9x makes some basic distinctions between types that can be extended and those that cannot.

The function of friend classes is one of the more controversial aspects of C++ (Schonberg, 1992). The built in privacy of class members that are not explicitly declared public means that it is impossible to write efficient code that makes use of distinct classes without the friend concept.

In Ada, the need for such a mechanism is lessened by the possibility of defining related types in the same package. Those types can be private and still have functions defined in the package that make use in their bodies of the private representation of these types. This style respects the interface between interface and implementation, but requires more design discipline. (Schonberg, 1992)

To maximize software reuse, it is important to be able to parametrize software components. The generic facility of Ada has type parameters with specified operations, both private and limited generic types. It is also possible to specify type parameters that belong to a given class of types as well as value parameters and object parameters. In Ada 9x it is possible to specify generic derived types (where the actual is any member of the class of the generic type) and generic formal packages (where the actual is any instantiation of the generic formal package) (Schonberg, 1992). This form of parametrization is more powerful than what is available in C++.

D. LANGUAGE CONCLUSIONS

To compare Smalltalk, C++ and Ada 9x is not easy. Smalltalk is a pure OOPL with simple syntax and semantics. It is best suited for the presentation of OO concepts and techniques, but does not contain the constructs for multitasking, large modular projects or real time programming. C++ has a large professional community which produces some exceptional code. Unfortunately C++ is also a language for which there is no stable definition, no approved standard reference and no translator validation suite (Schonberg, 1992). Ada code has strong standardization, and the resulting portability helps reduce software costs that have disproportionally grown as programs have increased in size. The "best" single language to incorporate into a OO curriculum will need to have characteristics of all the discussed languages. Smalltalk's clear OO concepts and simple semantics make it the perfect language for beginning students. C++ with its large backing and support structure require serious programmers to be competent in the family of C languages. Ada 9x brings to this group of languages the OOP capability with type extensions and real-time programming on top of its proven type safety and modularity. Smalltalk and Ada 9x have the qualities necessary for an OO curriculum, while the realities of the programming world demand the inclusion of C++.

IV. PIAGET'S DEVELOPMENTAL PSYCHOLOGY

One of the dominant figures in contemporary development psychology is Jean Piaget. This work provides a larger context in which to view the acquisition of knowledge. Piaget's ideas are concerned with the inter-play between logic and psychology as problems get solved. Piaget states that there are three main periods of psychological development in a child. Reviewing these developmental periods outlines the fundamental ability of a child to learn. I will review these periods and make analogies to them as they relate to the structuring of computer science curriculums.

A. SENSORIMOTOR INTELLIGENCE

Piaget states that knowledge begins at a base level from which all understanding must start. The example described is that of a new born child and its immediate "sucking reflex". This is the baby's base level of knowledge from which all learning will begin. The continual exposure to a situation allows acquisition of new knowledge. This "learning by doing" is fundamental in Piaget's form of development. Piaget describes how a child moves from blind repetition to repetition to make something last. Understanding requires access to the phenomena. Continual exposure to the new environment allows for familiarity and understanding. It also leads to active experimentation and inventiveness. The comprehension of new ideas all start from some individual basic level of knowledge. The child progresses from the "sucking reflex" to higher levels of understanding. The reflex is replaced by cries for food, then specific requests for an item and finally the ability

to actively determine "what's for dinner". Whether it be learning to walk or nuclear physics, understanding the environment is mandatory before complete comprehension can be obtained. Analogies can be made to the basic understanding of software systems and how through hands on experience, some basic system functions can be understood. This knowledge then leads to further discoveries.

B. REPRESENTATIVE INTELLIGENCE AND CONCRETE OPERATIONS

1. Pre-operational Phase

The description of the pre-operational phase begins with the ability to imitate in the form of images. One child initially is unable to see that the square peg will not fit into the round hole. Once images can be manipulated, symbolic thought is possible. The child sees the round whole and can imagine what the peg must look like in order to fit the hole. Image references with functional applications allow abstraction of thoughts and ideas.

One main idea from this period is the concept of conservation. When a child understands conservation he/she has the ability to understand that an object divided into parts is constant in its derived attributes. An example of this concept is the ability to understand that a pie cut in half contains the same combined weight and volume. Weight and volume are the derived attributes. This ability to divide general classes into logical sub-classes and maintain the derived attributes or their functional equivalent is the first step to understanding the concept of object oriented design.

2. Concrete Operations

The achievement of the early stage of conservation is the definition of what Piaget characterizes as concrete operations. A logical path has been followed by the child working his way through the logic of groups and the achievement of grouping structures. These grouping structures are the abstractions behind the child's concrete operational thought. The connection with computer science here is direct. The logical grouping of classes is integral to object oriented design. Understanding the causes and effects of actions is the basis of concrete operations. During the phase of concrete operations Piaget distinguishes the tendencies to seriate, to classify and to establish correspondence.

To seriate is to arrange objects in some sort of order. There is a developmental sequence in handling this task. For example, when arranging objects from smallest to largest size you obviously understand each object in the sequence is larger than the previous. Instinctively understanding the previous object is smaller than the current is also present. When successful the child or the student has mastered the logic of not only the direct relationship, but also its inverse.

To classify is to sort according to some quantity. Inclusive classes increase the difficulty of this process. In the previous example the objects can be sorted by color as well as length, increasing the situations complexity. This inclusive expression $(A+A')+B=A+(A'+B) = C$ shows that more than one kind of inclusive grouping can be made. The total remains the same regardless of the grouping. Additional difficulty arises

when requirements of similarity are included in the sort criteria. At the stage of concrete operations, to group in alternative ways is a higher order achievement.

Piaget describes another level that a child achieves in the grouping operation as correspondence. The simplest form of correspondence is the one-to-one correspondence in which each element of one set is placed into correspondence with an element of a second set. Piaget continues this discussion to describe the fit between a mathematical model and an empirical situation as a correspondence. To achieve correspondence in this situation it is sometimes necessary to bring into association more than one attribute. If objects are classified by color and shape, the relationship can be represented by multiplying the color classification (A1) by the shape classification (B1) producing the double classification (A1B1). There are many degrees of complexity with such matrices. These relationships can be extended many levels. The understanding of correspondence is the basis for understanding the concept of inheritance in OOP. The ability to understand this concept completes the second phase of concrete operations.

C. FORMAL OPERATIONS

The final period of development is that of formal operations. The thought process of the child moves from concrete operations to formal, propositional thinking. These operations are characterized as the scientific method, such as, consider all possibilities, make if-then hypotheses, organize the principle elements into some structure and come to a conclusion. Reasoning by hypothesis and a need for demonstration have replaced the simple stating of relations. Two important changes from the concrete operational

structures of seriation, classes and correspondences have taken place when the stage of formal operations has been reached. Instead of dealing with the concretely presented groupings, several of the grouping operations are combined, so a more generalized classification scheme is reached. In the formal operation stage, thought proceeds from a combination of possibilities, hypothesis and deductive reasoning, instead of being limited to deductions from the immediate situation. Additionally a new structure emerges that Piaget calls a four-group, representing identity, negation, reciprocal, and correlative transformations. The main characteristic of a system having a four-group structure is that it must contain two distinct and equivalent operations which have exactly equivalent outcomes. The ability to have numerous operations defined by there structure under the same name is a fundamental concept to be understood. Having reached the stage of formal operations, the development process has been completed.

The three distinct stages of development outlined in the section are applicable to learning any new paradigm. Developing a computer science curriculum around the natural phases of knowledge acquisition can ease the transition for structured programmers and provide a logical foundation for beginning computer science students.

D. MAPPING PIAGET'S DEVELOPMENTAL PSYCHOLOGY TO TEACHING OOP

In order to map an object oriented language with developmental psychology, I will clarify the different roles a modeling process plays in the programming process. The programming process can be described as a modeling process in which several

sub-processes take place. Figure 1 illustrates the programming process as a modeling process between a real world system and a model system. For this discussion a phenomenon is something that has definite existence. The real world system is part of the world that is being focused on in the programming process. The model system is a program modeling part of the real world system on a computer. The real world system

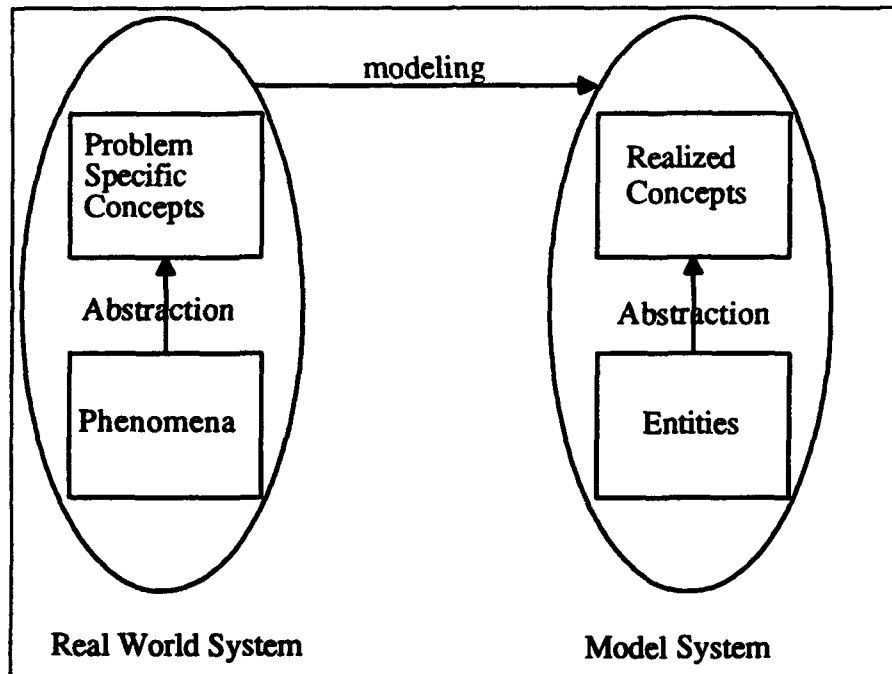


Figure 1. Modeling Process

consists of phenomena from the physical world and concepts used to capture the complex world. Both phenomena and concepts are important in the real world system. As an example, Smalltalk objects are usually models of physical phenomena in the real world system. Concepts are modeled by abstractions such as classes and methods. The program text can be thought of as a description of the real world system. Again referring to Figure 1, during the programming process there are three sub-processes: abstraction in

the real world system, abstraction in the modeling system and modeling. In this case, abstraction in the real world system is the process of perceiving and structuring knowledge about phenomena. This process creates concepts that are problem specific. Abstraction in the model system is the process to build support structures that are intended to be created on a computer. Realized concepts are created in the model system. The modeling process is the connection of problem specific concepts to realized concepts. (Knudsen and Madsen, 1988) This connection is the critical skill required to progress from Piaget's concrete stage to the area of formal operations.

1. Mapping

In this section an outline of a mapping from Piaget's three developmental stages to the corresponding OO concept and then to a recommended programming example is made. By following this outline the connection from problem specific concept to realized concept can be produced. Piaget's first development stage is sensorimotor intelligence. Row #1 in Figure 2 can be defined as the level of empirical concreteness. Students do not realize similarities between different phenomena, nor do they obtain any systematic understanding of the individual phenomena. Students notice what happens, but do not understand why it happens or the different relations affected. In the programming process this corresponds to a level where the students are trying to understand the single objects that constitute the system. There is little understanding between the relations of the objects or how to group them into classes. I correlate this stage to the basic understanding of the operating environment. Further exploration cannot begin until an understanding of the surroundings has been completed.

	OBJECT-ORIENTED CONCEPT	PROGRAMMING EXAMPLES
Row #1 SENSORIMOTOR INTELLIGENCE	Basic System For/While Loop	Programming Environment
Row #2 REPESNTATIVE INTELLIGENCE AND CONCRETE OPERATIONS	Object->Message Classes Inheritance Polymorphism	Programming Language Issues
Row #3 FORMAL OPERATIONS	Abstraction Classification Aggregation Generalization Scientific Method Precondition Postcondition Loop Invariant	Problem Solving Program Design

Figure 2. Mapping Piaget to OO Concepts

Fundamental programming techniques are also required before advanced concepts can be attempted by the beginning student. Basic problem solving algorithms should be presented in this stage to develop a confidence in the initial techniques. A sample program should be developed to demonstrate the environment and some initial programming constructs.

Piaget's second development stage is representative intelligence and concrete operations. Row #2 is defined as the level of abstraction. To understand the complications of the real world system the phenomena must be analyzed and concepts developed for grasping the actual properties. This corresponds to designing the classes

level of abstraction a simple and systematic understanding of the phenomena in the real world system is obtained. This section of the curriculum involves the introduction of the fundamental components of object-oriented design. Classes, messages, methods, inheritance and polymorphism are presented as examples with clear purpose and applicability. The sample program developed in stage one can be extended as a problem statement to include components of object-oriented design. The OO methodology is demonstrated as a problem solving technique as the necessary structures are built.

Piaget's third developmental stage is formal operations. Row #3 is defined as the level of thought-concreteness. The understanding corresponding to the abstract level is further developed to obtain an understanding of the total real world system. By organizing the phenomena of the real world system by the concepts established, the ability to understand the relations between the phenomena becomes available which was not at the level of representative intelligence. In this stage the conceptual theories are presented on the foundations of the concrete operations. Formal operations are mapped to OO concepts in two sub-groups. The ability to perform abstractions through classification, aggregations or generalizations are developed as the theories are presented to the students. Understanding of the scientific method completes the third phase as the ability to examine abstract possibilities, create hypothesis and reason problems deductively are continually demonstrated and refined. The ability to explain why things happen and predict what will happen is present.

The process of creating new concepts is not entirely comprised of abstraction and its sub-processes; classification, aggregation, and generalization. Generally the concept definitions evolve through many changes. The understanding gained during development will influence further steps. Being aware that the concept of abstraction is composed of sub-processes is useful to the same extent as understanding whether a problem is approached top-down or bottom-up. The difficulty of this section is in understanding that the concept itself is an abstraction. Given a number of concepts it is possible to describe their structure in terms of classification, aggregation and generalization. It is important for students to be aware of this distinction. The ideas, examples and descriptions in this chapter are attributed to the writings of Jean Piaget. (Hilgard and Bower, 1975)

V. APPROACHES TO TEACHING OOP

The OOP technique helps programmers master the problems of complexity and is particularly powerful when the programs are large. The OOP paradigm is increasingly included in the undergraduate curriculum, but in most cases it is presented as just another language. This greatly limits the OO methodology and does not allow demonstration of its main strengths of code reuse, data abstraction and encapsulation. This chapter will discuss some of the basic principles of instruction and how they pertain to the computer science curriculum. The principles presented in Jean Piaget's *Developmental Psychology* discussed the developmental periods that outline a child's ability to learn. These principles are seen throughout this chapter and provide clarifying explanations for the proposed instructional method.

A. PRINCIPLES OF INSTRUCTIONAL DESIGN

Instruction is a human process whose purpose is to help people learn. In this section reviews what characteristics instruction must have in order to be successful and outline an instructional design approach that is both feasible and worthwhile. "Our research suggests that the knowledge of novices is organized around the literal objects explicitly given in a problem statement. Experts' knowledge, on the other hand, is organized around principles and abstractions that subsume these objects." (Glaser, 1984)

There are alternative ways in which to design the instruction of individuals and individual subjects. This section describes one way that is both feasible and worthwhile.

It is a general approach not limited to the computer science field. There are five assumed characteristics that need to be mentioned.

1. Aid the Individual

Instructional Design must be directed at the learning of the individual. The concern is not with the mass changes in opinions or abilities, but with that of the individual student. Even among an assembled group the learning must occur within each individual. This does not mean customized instruction. Rather it is an attempt to ensure all students are meeting standards required for further development.

2. Short and Long Term Outlook

Both immediate and long-range phases are part of instructional design. The immediate phase is in the form of daily objectives and the individual concepts that are required to be completed. The long-range phase will consist of a set of lessons organized into topics; a set of topics consisting in a course; and a sequence of courses that encompass the entire instructional system. The immediate phase design responsibilities are typically controlled by the class instructors while long-range designs are undertaken by groups of scholars representing academic disciplines.

3. Systematic Instruction

Instruction designed systematically will have a positive affect on individual development. Unplanned or undirected learning leads to diffused understanding of basic concepts. This leads to uncertain expectations as careers progress and higher level classes attempt to build on previous knowledge.

4. Systems Approach

Instruction design should be conducted by means of a systems approach. The systems approach to instructional design involves the completion of steps beginning with an analysis of the needs and goals and ending with an evaluated system of instruction which succeeds in meeting the accepted goals. Decisions in each step are based on empirical evidence. The process is based on human reasoning with each step becoming an input to the next step.

5. Developmental Psychology

Designed instruction must be based on knowledge of how human beings learn. When attempting to develop an individual's ability it is not enough to state what those abilities should be. The question should be asked, how are these abilities acquired? Instructional design must take into account the learning conditions that must be established in order for the desired effects to occur. (Gagne and Briggs, 1979)

B. INSTRUCTIONAL SYSTEM DEVELOPMENT

The process of developing an instructional system is accomplished as a series of stages. Each stage is continually modified as insights are gained from new discoveries. The design effort is divided into three sections; System Level, Course Level and Lesson Level (Gagne and Briggs, 1979). The design makes use of all theory and research evidence available while being supplemented by the iterative self correcting process of empirical tryout and revision.

1. System Level

The system level will determine in a general sense the distance a instructional system needs to travel in order to produce the desired abilities in the students. An analysis of the abilities of incoming students, a list of objectives attainable by outgoing students and an order of importance is associated with each area of concern. Time available with the students and the resources allocated constrain the scope of training and limit the realistic goals. It is not enough to define the goals that are to be attained. The correct path to accomplish these goals is also required. What is the best way to learn this subject? From whom or what approach is the most efficient method of delivery?

2. Course Level

The system level has defined the major skills to be learned during the course of the curriculum. In order to accomplish these broad skills target objectives need to be defined. This step considers the sequencing of major clusters of course objectives for each year in the curriculum. These clusters are defined as "units of instruction" (Gagne and Briggs, 1983). Having grouped the course target objectives in some fashion, a loose structure is formed. Completion of these units of instruction satisfy the target objectives.

Working from the general needs and goals stated in the systems level, to the more specific course objectives, often produces improved ways to organize the instruction. This iterative cycle to the process of instructional design is continued as specific content is added to the system.

There is profit in undertaking three kinds of analysis of objectives at this point:

(a) information processing analysis, to reveal the sequence of mental operations in performance of the objective, (b) task classification, to categorize type of learning outcomes in order to identify the conditions of learning, and (c) learning task analysis, to reveal the enabling objectives for which teaching sequence decisions need to be made (Gagne, 1977).

3. Lesson Level

Systematic development and review has been given to the needs and goals first specified in each course in the curriculum scope and sequence statement. Formulation of the results of the various analysis into curriculum purpose, course objectives, unit objectives, target objectives and finally enabling objectives provide a clear picture of the path necessary to reach the stated goals.

The instructional system has been defined with the overall design process as a set of stages for analysis and development. The systems level focused on the determination of needs and goals sought as outcomes from the curriculum. The goals are broadly stated and arranged as desired outcomes. The next stage determines the major units of instruction and a listing of the objectives to be achieved. This area is described as course level analysis. The lesson level incorporates the definition of detailed performance objectives, lesson plans, selecting course materials and preparing measures for assessing student performance.

C. EVOLUTION OF COMPUTER SCIENCE CURRICULUM

Computer science has evolved through distinct curricular approaches since the 1960's. Before proposing an alternate approach, this section will review the historical evolution of introductory computer science curricula, including Curriculum 69 and 78, the Liberal Arts Model Curriculum of 1986, the Denning committee's comprehensive approach in 1989, and the ACM/IEEE report, Computing Curricula 91.

1. Curriculum 68

In the 1960's, computer science emerged as a distinct discipline. In order to define the scope of this new discipline Curriculum 68 was proposed by the ACM Curriculum Committee on Computer Science (Communications of ACM, 1968). Curriculum 68 organized computer science into three sub-fields: information structures and processes, information processing systems, and methodologies. Curriculum 68 proposed a core curriculum of four basic courses (algorithms and programming, computer and system structure, discrete structures and numerical calculus). This is followed by four intermediate courses (data structures, programming languages, computer organization and system programming. Curriculum 68 emphasizes numerical analysis and hardware more than the current core curriculum but omits software engineering (Communications of ACM, 1968).

2. Curriculum 78

Curriculum 78 refines the previous work to suggest that the emphasis should be placed in algorithms, programming, data structures, and hardware. It also includes a list of topics and fundamental knowledge that every computer science major should know,

and suggests that every major should obtain the following six skills: the ability to write programs, measure the efficiency of programs, understand the problems that are applicable to computer solutions, understand problem solving and be prepared to pursue graduate study in computer science. (Communications of ACM, 1979)

Introductory courses CS1 and CS2 of Curriculum 78 were revised in 1984. (CS1 and CS2 are terms used to represent first and second year CS courses respectively) The objectives for CS1 became: to introduce a disciplined approach to problem solving, to introduce procedural and data abstraction, to teach good programming style, to teach a block structured language, and to provide familiarity with evolution of computer hardware and software. CS2 had the following objectives: to continue developing a disciplined approach to programming, to teach data abstraction and data structures, to introduce different implementation strategies for data structures and to introduce searching and sorting algorithms and their analysis. (Tucker and Wegner, 1994)

3. The Liberal Arts Model Curriculum

This alternative approach presented in 1986 emphasizes that computer science has a coherent body of scientific principles. It defines computer science as the systematic study of formal properties, implementation, and application of algorithms and data structures. The Liberal Arts Model for the CS1 courses follows Curriculum 78. The second year places greater emphasis on conceptual and formal tools. Its goals include: to consolidate the knowledge of algorithm design and programming emphasizing the design and implementation of large programs, to begin a detailed study of data structures and

data abstraction as exemplified by packages or modules, to introduce mathematical tools such as complexity and program verification and to provide an overview of the rest of computer science including computability and architecture. (Gibbs and Tucker, 1986)

The Liberal Arts Model proposed CS1 follow the curriculum outlined in Curriculum 78, while its proposal for CS2 places greater emphasis on conceptual and formal tools (Tucker and Wegner, 1994).

4. Denning Report

The 1989 report of the Core Curriculum Task Force of the ACM reexamined the scope of computer science, proposed a new teaching paradigm, and presented an example of an introductory course sequence. It defines the discipline of computing as "the systematic study of algorithmic processes that describe and transform information: their theory, analysis, design, efficiency, implementation and application." (Denning, 1989) It divides the discipline into nine sub-areas: 1) algorithms and data structures, 2) programming languages, 3) architecture, 4) numerical and symbolic computation, 5) operating systems, 6) software methodology and engineering, 7) database and information retrieval, 8) artificial intelligence and robotics, 9) human-computer communication (Denning, 1989). This report proposes a three semester introductory sequence to cover all these areas. This sequence is less oriented to programming than earlier approaches. It has a broader overview of the discipline and is geared toward the CS major.

5. Curriculum 91

The ACM/IEEE report Computing Curricula 1991 defines the definition of the computing discipline for undergraduate curriculum. It is a design document for curriculum rather than a pre-designed curriculum. Curriculum 91 intentionally encourages curriculum innovation at the introductory level. Curriculum 91 makes the statement that programming is pervasive and that students should receive training in problem solving and programming early and often in their undergraduate course work. The report does not answer questions like, "What languages?" "What programming paradigms?" and "What kind of programming?" This report intentionally leaves open the possibility of an OO approach. Curriculum 91 makes statements about each of the design issues; the need for breadth of discipline coverage, the role of programming; the nature and role of labs; interaction among the processes of theory, abstraction, and design; and the need to address social and professional issues. The breadth of the discipline coverage is ensured in the reports recommendations in the form of "knowledge units" (Tucker and Wegner, 1994). These topics cover the nine major areas of the discipline as defined in the Denning Report.

D. APPROACHES TO CURRICULUM DEVELOPMENT

A curriculum can be designed with different areas of emphasis. These areas are shown in Figure 3 (Tucker and Wegner, 1994) as three dimensions. The three dimensions are the "Level of Abstraction", "Subject Coverage" and "Learning Style".

The Level of Abstraction is the dimension between theory and practice. Theory in a course includes reasoning, formal methods, attention to concepts and algorithms. Practice would include activities like modeling, informal methods, relevant examples and the demonstration of user interfaces. The mixing of theory and practice is contained in many courses, although the amount of each can vary greatly. A CS1 course may consist of only programming instruction or a more abstract concept like problem solving.

A second dimension in curriculum design is that of Subject Coverage. This is the option to cover a single topic in a course or choose two or more major topics. The choice is between the depth or breadth of topic coverage. A programming course in CS1 could include OO design principles and user interfaces as additional topics.

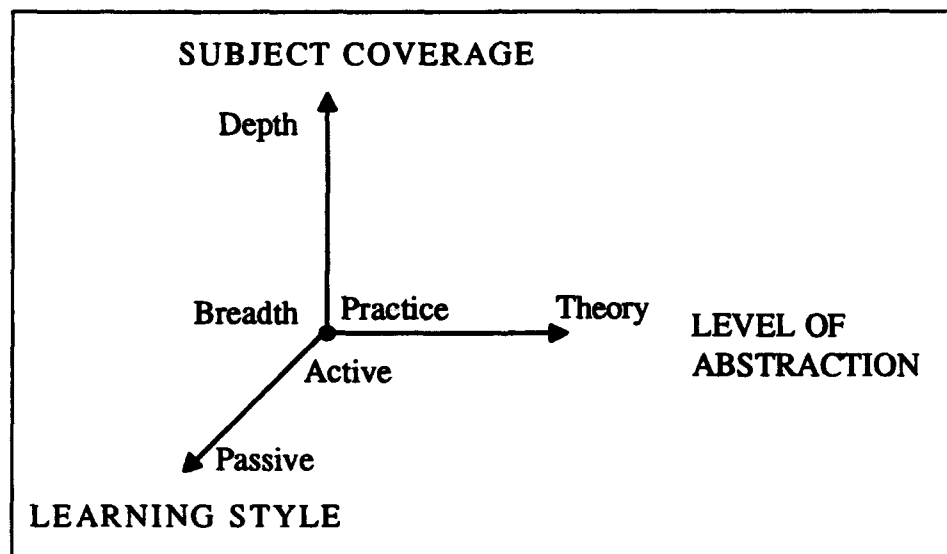


Figure 3. Dimensions in Curriculum Design

Learning Style is the third dimension in curriculum design and can range from completely active to completely passive. Active learning would include the actual

programming assignment, analyzing the program or presenting computation findings. Passive learning involves lectures and assigned readings.

E. APPROACHES TO TEACHING OOP

This thesis advocates teaching OOP in CS1. While the OO technique is not a recent invention, it has only recently reached a point where the need for a replacement to the structured paradigm has been recognized. Advantages of OOP are that the modern languages more fully support abstraction, making it easier to write reusable code. They also support top-down programming and closely resemble the way in which humans solve problems.

Within the OO community, two schools of thought are apparent. The first believes that OOP should be taught after an introduction using a more traditional approach. The belief is that jumping into objects at first is too much for novices since new objects must be created before anything else can take place. Starting off with memory management issues is too difficult. (Wu, 1993)

The second view is that moving from a procedural paradigm to an object-oriented paradigm in one semester is too much for introductory students. This approach introduces students to the pure OO approach from the beginning. Students have a clear understanding of the paradigm before transferring their skills to other languages.

Both schools agree that the environments used to teach OOP should include tools that support OOP. It is important to provide students a complete understanding of the OO paradigm before exposing them to the detail of hybrid procedural/OO languages such

as C++. At the Educator's Symposium at OOPSLA 1993, there was a consensus that even if a course uses the hybrid C++, a more "pure" OO language such as Smalltalk should be used first, to give a clean introduction to OOP.

There are numerous approaches available once the OO paradigm has been selected as the proper curriculum. This section will highlight the advantages and disadvantages associated with each approach.

1. Top-down/Bottom-up

The top-down approach presents the conceptual aspects of the OO languages first. The main concepts of objects, classes, abstraction, inheritance, and encapsulation are explored. This is followed by the specific implementation concepts of the chosen language. The top-down approach is beneficial to students that already have an understanding of basic programming. For the beginning student the top-down approach may be too abstract (Wu, 1993).

The bottom-up approach begins with the details of a language and how to solve minor problems. Step by step capabilities are added and problems are expanded. In this way the concepts of the OO paradigm are presented with examples and applications. This approach is more suitable to the beginning student (Wu, 1993). Executable programs are created by the students from the beginning providing a understanding of not only the language, but also the operating system and the language concepts. The limitations of the bottom-up approach include the initial learning period required for

beginning students to understand the language. Also the equating of OOP with the language on which the concepts are learned may limit future exploration.

2. Pure/Hybrid Languages

OOPL's may be pure OO, or they may be one of the hybrid languages. A pure OOPL (Smalltalk) treats everything in a program as an object. The creation or use of existing objects places additional requirements on the beginning programmer. The syntax and semantics of the language must be understood in addition to the concepts of OOP (Wu, 1993). The incremental learning process that makes the bottom-up approach desirable is hard to implement with a pure OOP. There are many concepts and skills required to be able to write even a simple program with a pure OOPL. A hybrid OOPL (C++) eliminates the object overhead required with a pure OOPL. However, the concept of an object is not a requirement a hybrid language and can therefore be avoided or used improperly. The writing of code in C++ does not mean the code is object-oriented. This is confusing to some programmers.

F. DESIGN CONCLUSIONS

In this chapter the discussion has covered the principles of instructional design in which basic teaching assumptions and system development are explored. The main points from this chapter when developing an OO instructional system are:

1. Curriculum 91 from the ACM/IEEE report defines the computer science undergraduate curriculum without specifying a programming paradigm. An OO curriculum may be based on this report and satisfy the guidelines of the ACM/IEEE.

2. Design the instruction to aid the student with incremental progressions in understanding. Piaget's three stages of learning are applicable to teaching programming.

3. System goals, course structure and lesson objectives require understanding of both long and short term requirements. The short term requirements of the system must quickly get the student involved with and understand the potential benefits of the new approach. The long term must include a language that is powerful enough to be applied to real applications.

VI. PROPOSED CURRICULUM

For several years calls have been made to incorporate the OO methodology into the CS curriculum. (Temte, 1991) The claim has been made that object-oriented technology will become the dominant software development methodology, replacing the traditional functional decomposition model. (Lutz, 1990) This chapter will propose a CS1 curriculum to incorporate this methodology while addressing the issues raised in Chapter IV and V concerning the way people learn. The proposed curriculum will encompass the first year only. The curriculum will look at System and Course level topics. Redefining specific areas of study is not the purpose of this thesis. The preparation for advance CS topics or the various disciplines utilizing computers is the concern. The proper curriculum structure supported by the advantages of object-orientation will provide students with a strong foundation regardless of the advanced path chosen.

A. INTRODUCTION

The first year Computer Science curriculum (CS1) is increasingly being challenged to lay the foundation that will produce graduates to satisfy the needs of business, industry and graduate programs. The demands placed on CS1 to educate CS majors are often in conflict with the increasing number of programs requiring computing education. These challenges have not gone unnoticed (Denning Report, Curriculum 91). This thesis has looked at aspects of learning as described by developmental psychologist and instructional designers. The persons quoted differ in background and training, so it is not surprising

that their approaches differ. Piaget's developmental psychology is a bottom-up approach beginning at some base level and moving to concrete operations and formal propositional thinking. The instructional design process for system development is a top-down approach. Each approach has its advantages and can be used together to produce a curriculum that meets the needs of all students of computing. This chapter will look at the need for computer science curricula reform, the foundations of computing, the integration of these two different approaches to learning and finally the proposed curriculum.

B. SYSTEM LEVEL REFORM

The revision of a CS curriculum on the single basis of switching to JO principles would be short sighted and fail to address all that needs to change. Simply changing programming paradigms will not fix the problem. A complete look at all aspects of CS education is required.

1. Standard Curricula Model

The standard model for baccalaureate study is based on two parts that specifies a lower division and upper division with each having a distinct and identifiable mission. (Shaw, 1984) The lower division's mission is to establish a foundation in the subject matter that is broadly applicable to computer science. The upper division should focus on particular areas that take advantage of the students broad foundations and enhance these abilities to apply concepts, techniques and problem solving approaches. Both the Denning Report and Curriculum 91 confirm the generic model of the baccalaureate curricula.

There is interest in broadening the role of the lower division courses to include interdisciplinary integration of concepts and skills.

2. Computer Science Curricula

The traditional computer science curriculums do not conform to this model. The lower division focuses on skills training and leaves the conceptual material to be introduced in the upper division or largely ignored.

Traditional CS curricula reflect an implicit dichotomy quite different from that of the rest of academia: lower division programs in what might be described as "Basics of Programming Skills" and upper division programs in what might be called "Foundations of Computer Science". (Shackelford and Leblanc, 1994)

The main curricular problem is the foundation of computing as a discipline is withheld from CS majors until their habits and biases are already established. And if only a few CS courses are taken (non CS majors) then the foundations of computing are completely missed. The first few courses in the CS curriculum functions in a bottom-up approach which works well as a training tool but fails in the long term goal of education. Computing has matured to the point where its baccalaureate model requires fundamental revision and the required revision must include a focus on the intellectual foundations of computing in the early courses.

C. COURSE LEVEL REFORM

When proposing a CS curriculum it is not appropriate to look at only the language methodology. A complete perspective is required beginning at the lowest level and working up through the various issues. Computing is at the center of virtually every advance in human knowledge. At the center of computing is the algorithm. Algorithms are no longer strictly in the domain of computer scientists. They now belong as the foundation for computing professionals from virtually every field. The algorithm is at the center of this computer revolution and so it must be at the center of the CS curriculum. The curriculum recommendations made in Curriculum 91 and Denning recognize the algorithm as a key factor and favor the integration of theory, abstraction, design and experimentation.

This integration should be a central feature of the beginning algorithms and programming courses since they must provide undergraduates with:

- the first exposure to the fundamental ideas of algorithms.
- the first disciplined introduction to abstraction and design.
- exposure to technologies and practices of design and implementation.
- early exposure to systematic experimentation.
- an early exposure to algorithmic problem solving. (NSF, 1989)

Algorithms and programming courses are becoming foundational for a growing number of computing specialists from many other disciplines. (Foley and Standish, 1989) The enrollment of non-CS majors in the algorithms and programming courses is large, is rapidly growing and has been identified as an accelerating trend. (Shaw, 1984) For computer science departments this means their curricula must feature lower level

algorithm and programming courses that serve both CS students and students from other curriculum. The teaching of traditional programming (algorithms, syntax and semantics) is still required regardless of the final methodology.

D. INTEGRATING APPROACHES

1. Bottom-up and Top-down

Typical CS1 curricula feature programming instruction in a bottom-up approach. Upper level courses are the domain of big picture subjects and use a top-down approach to teaching. This "Programming Skills First", "CS Principles Last" concept results in the weakness seen in the CS curriculum.

Prior to implementing the new curriculum, our conclusion of software engineering principles in traditional introductory programming courses indicated that students were capable of learning the skills but did not incorporate them into their habits. In retrospect, this is not surprising: we were insisting that students follow practices that had no positive value to them. (Shackelford and LeBlanc, 1994)

The traditional curriculum divides CS education into the treatment of programming skills and conceptual foundations. By leaving out these foundations the student perceives programming skills as arbitrary instructions with little or no meaning. To understand the importance of complexity management both programming skills and the conceptual knowledge must be presented together. Instructions for the simple reason of "because I said so" work no better now than when we heard them as children. For students to

combined the areas of theory, abstraction, design and experimentation the CS1 curriculum must integrate the presentation of programming skills and conceptual foundations.

2. Inclusion of CS and Non CS Students in CS1

CS is increasingly important to all education fields. CS1 must provide the tools necessary for all students to explore a wide range of complex phenomena. Any discipline that utilizes computers will eventually encounter the need for algorithmic models. A set of introductory courses that provides strictly programming instruction is not sufficient for today's non CS students.

E. PROPOSED CURRICULUM

Any curriculum for today's variety of CS students should provide some hands on experience that demonstrates the benefits of appropriate programming practices and the negative results that occur when these practices are neglected. Additionally, the conceptual knowledge these practices are designed to manage should be presented concurrently to provide realistic meaning to the programming instruction. The top-down approach will provide coverage of the conceptual foundations of computing, which includes the structural properties of algorithms. The bottom-up approach will provide the demonstration of applications, programming and beginning problem solving.

1. A Model For CS1

The model for the CS1 curriculum features a set of courses composed of two levels and five courses total. Level 1 features two first quarter courses, "Introduction to Computing" and "Introduction to Programming" which should be taken close together

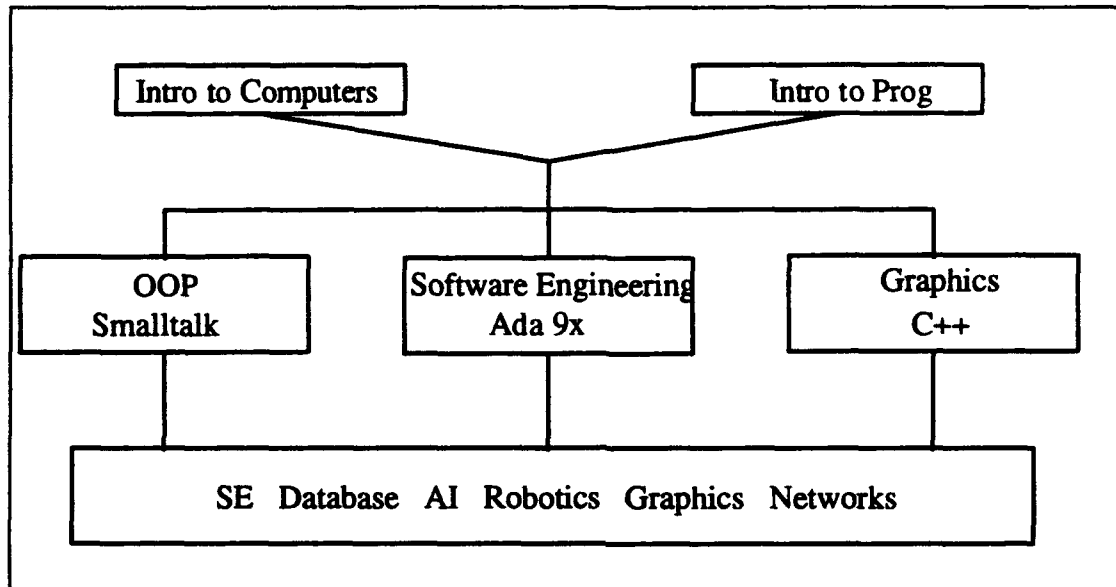


Figure 4. Curriculum Model

and are designed for both CS and non CS majors. Level 2 is a set of three 2nd quarter courses, each covering foundational CS material.

Figure 4 displays the proposed model which begins with two introductory courses. The introductory courses are both required but the sequence is irrelevant. For every argument proclaiming which course should be first an equaling compelling argument can be presented for the opposite sequence. What is important is that the material is presented as close together as possible, preferably concurrent. The level 2 courses are intermediate courses that begin the specialization process. For CS students, parallel

studies that combine the benefits of each intermediate area are recommended if not mandatory. Level 2 courses can be tailored by the upper level curriculum requirements.

What is important is that the foundations are in place from which students can build.

a. Introduction To Computing

The intended purpose for the Introduction to Computing course is to:

- Present the algorithmic model in the context of technology in modern life. Examples and applications of the impact of computing on the areas of natural science, engineering and business will provide an understanding of where computing has been and some of its accomplishments.
- Provide an introduction to the concept of computing.
- Demonstrate to students, from communication packages to design toolkits, the range of applications available today to assist in controlling technology.
- Experience the design and implementation of pseudo-code algorithms and data structures and initiate the integration of analysis, abstraction design and OO modeling.

This course is designed to remove the programming implementation details which usually cause students to become immersed in debugging and execution of the code, instead of understanding the analysis and design of the problem. Students are allowed to focus on the conceptual issues that are of real importance and not fight with the unfathomable complexities of the compiler. The Introduction to Computing will provide concepts and applications of computers which can be made as challenging as necessary to interest students with novice to advanced skills.

b. Introduction To Programming

The assumption made in the design of this course is that students have little or no computer experience. Beginning at the "beginning" ensures everyone develops as the curriculum design intends. The approach for this course is based on the series of articles by (Wu, 1993). This course is divided into three major stages.

1) Stage 1. Non-OOP

In this initial stage the mechanics of the language (For NPS, I recommend Ada) are explained and illustrated. The concepts of object-orientation are not yet discussed. A sample program (Sample Program 1) is begun which will be used throughout the course to bring continuity and demonstrate the advantages of incorporating OO concepts. This approach will provide students with an appreciation of how and why the OO method helps produce better programs. The objectives of the first stage are to:

- Make students become proficient enough in the chosen language to be able to write a simple program and teach them how to use the environment.
- Wean intermediate students from the old way of thinking.
- Inform students that traditional programming is difficult and not appropriate for developing large programs.
- Introduce students to a sample program that will be expanded into a more complete and robust program in the following two stages. (Wu, 1993)

2) Stage 2. Semi-OOP

The concepts of abstraction, encapsulation and polymorphism are introduced in this stage as Sample Program 1 is extended. Wu recommends the treatment of objects as black boxes whose functions are explained in the next stage. Emphasis is placed on utilizing the behavior of the classes and demonstrating the advantages of code reuse. Object creation is introduced in the next stage. The objectives for the second stage are to:

- Introduce the benefits of code reuse.
- Describe the notion of abstraction and encapsulation.
- Discuss the shortcomings of client programming. (Wu, 1993)

3) Stage 3. Full OOP

The full concepts of OOP are introduced in this stage. The emphasis is placed on how to create programmer defined objects. Class and inheritance are explained and illustrated. Sample Program 2 is used as a reference to show its weakness and limitations. The Sample Program 2 is improved by creating programmer defined objects. The OO design guidelines in which objects are classified into four categories is introduced in this stage. The four categories: Interface, Control/Computation, Data Management, and Application are intended for be an informal guide for beginners to utilize in designing programs. The objectives of the third stage are to:

- Introduce server programming.
- Introduce the concept of inheritance and polymorphism.
- Reemphasize the importance of abstraction and encapsulation.
- Introduce an OO design methodology.
- Lay the foundation for advanced study. (Wu, 1993)

For additional details refer to (Wu, 1993)

c. Intermediate Courses

Level 2 courses are intermediate courses that combine conceptual study and applied applications in the foundations of computer science with a concentration in a particular programming paradigm. Each independent course will reinforce the fundamental material presented in Level 1, introduce new programming paradigms, languages and problems, and provide additional foundations for applications to be presented in upper level computing topics. The Level 2 courses combine applied work and study in data structures, algorithm analysis, software design and software engineering principles with a particular programming paradigm. The example in Figure 4 illustrates a

curriculum which has three Level 2 courses: OOP with Smalltalk, Software Engineering with Ada 9x, and Graphics with C++. Each course is able to introduce students to a particular application domain while providing coverage of foundational computer science material. Level 2 courses serve three primary missions: introduce different programming paradigms, teach fundamental CS subjects and lay foundations for advanced study.

The main question regarding intermediate courses is, how well will relatively inexperienced students respond to a variety of paradigms and programming environments?

Younger students adapt more quickly than do upper classmen and graduate students. It appears that the relative lack of programming experience proved to be an advantage in adapting to new paradigms. (Shackelford and LeBlanc, 1994)

F. CONCLUSIONS

This curriculum will introduce the foundational concepts and techniques beginning at the introductory level. This is a correction of the "Depth Last" model most common in CS curriculums. Instead of solely providing training in programming skills at the introductory level, this curriculum presents education in conceptual foundations. There is integration of theory, abstraction, design and implementation. Material presented in this fashion will clarify proper computing techniques.

In the continually transforming field of CS, students must be prepared to adapt to new environments. In the past, students trained on a single platform with a single language. Students experienced with a single programming language are less open to

change and are less adaptable than students exposed to various platforms and languages.

Upper division courses can focus on their proper direction with a clearer picture of the foundations attained by the students. By introducing the fundamental material early in the curriculum advanced courses can devote more effort within their respective fields.

This curriculum incorporates students from all disciplines and provides the same curricular structure. The CS1 curriculum combines the applicable skills and techniques with the conceptual understanding necessary for students to be proficient in using computers. Non CS students have the abilities needed for further course work. CS students are fundamentally sound and ready for additional study.

This model for CS1 addresses the various Curriculum challenges that effect both CS and non CS majors. Through the integration of the top-down and bottom-up approaches, it provides flexibility for all students, covers foundational issues early and provides continuity from which advanced courses can build.

VII. CONCLUSIONS

The OO paradigm is a powerful tool for modeling real world applications. It is the coming together of many different concepts all applied to the task of programming. OOP is the natural step in the evolution of higher order programming languages. This thesis looked at the main principles involved in OOP and reviewed three main languages. The fundamentals of learning theory were developed as a basis for the proposed CS1 curriculum.

A. OO CONCEPTS AND LANGUAGES

While many CS professionals acknowledge the potential value of the OO paradigm, many still regard it as an advanced topic suitable only for upper level students. This thesis looked at the fundamentals of OOP and found a well constructed set of concepts. As business and industry have recognized, the advantages of modular program construction are vital to the production of large scale systems. The software concepts that are stressed in structured programming become tangible and meaningful when presented in context with OOP. Structured programming is approximately twenty years old. Industry has seen the advantages of the OO paradigm. It is essential for academia to begin introducing these ideas at the earliest levels.

B. THE PSYCHOLOGY OF LEARNING

Humans learn in a manner comprised of basic understanding, concept manipulation and propositional thinking. Instructional systems that are developed in this manner provide a natural mechanism for comprehension. Understanding the basics of computer systems, algorithms and problem solving are fundamental. They can be presented without the confusion of language semantics or operating systems.

Humans naturally think in terms of objects. These concepts, presented initially in the CS1 curriculum, give meaning to the principles of proper software engineering. The unlearning of structured programming is not necessary when the OO concepts are presented from the onset. The paradigm shift is difficult only in one direction, from procedural to OOP. Using OOP as a means for developing procedural programming skills is much easier because it reflects how the paradigms are meant to fit together. Classes can provide a context for functions.

With a solid understanding of the basics and concepts of the paradigm the "leap" to abstract thought proceeds naturally. Generalization and abstraction are consistent with the way humans manipulate system understanding. The foundations previously presented flow into the abstract methods involved with problem solving.

C. OO INTEGRATION INTO THE CS1 CURRICULUM

The proposed curriculum demonstrates there is a feasible method for restructuring CS1. A refresher quarter provides the time necessary to prepare students with the courses, Intro to Computers and Intro to Programming. Intermediate courses provide individual areas of study and develop skills necessary to advanced subjects. All areas of study will benefit from a consistent presentation of the proper methods to utilize computers. This thesis has demonstrated the feasibility of OO integration into the CS1 curriculum.

LIST OF REFERENCES

ACM Curriculum Committee on Computer Science, "Curriculum 68-Recommendations for the Undergraduate Program in Computer Science", Communications of the ACM 11, 3 (March 1968), 151-197.

ACM Curriculum Committee on Computer Science, "Curriculum 78-Recommendations for the Undergraduate Program in Computer Science", Communications of the ACM 22, 3 (March 1979), 147-166.

ACM/IEEE-CS Joint Curriculum Task Force, "Computing Curricula 1991", ACM Press and IEEE-CS Press, New York, 1991.

AMSR, Intermetrics, "Ada 9x Mapping Specification and Rationale", version 4.1, Boston, Intermetrics, March 1992.

Anderson, C. "Ada 9x Project Report to the Public", CrossTalk, October 1992.

Bertino, E. Martino, L. Management Systems Concepts and Issues. Prentice Hall International Ltd. 1991.

Berzins, V and Luqi, Software Engineering with Abstractions, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

Booch, G. Object-Oriented Development, IEEE, New York, 1986.

Booch, G., Software Engineering with Ada, The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1987.

Booch, G., Object-Oriented Design with Applications, The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1991.

Budd, T., An Introduction to Object-Oriented Programming, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

Chorafas, D. Steinmann, H. Object-Oriented Databases. Prentice Hall, New Jersey, 1993.

Coad, P. and Yourdon, E., Object-Oriented Design, Yourdon Press, Englewood Cliffs, New Jersey, 1991.

Denning, P., D. Comer, D. Gries, M. Mulder, A. Tucker, A. Turner, and P. Young, "Report of the ACM Task Force on the Core of Computer Science", ACM Press, New York (1988).

Denning, P.;et. al., "Computing as a Discipline. (Final Report of the ACM Task Force on the Core of Computer Science)", Communications of the ACM, v32 p9(15) Jan 1989

Digitalk, Smalltalk/VWindows Object-Oriented Programming System, Digitalk, Inc, Los Angeles, California, 1991.

Fairley, R. Software Engineering Concepts, McGraw-Hill, New York, 1985.

Foley, J. and Standish, T. "Report of the Computer Science Workshop", Undergraduate Computer Science Education, Section 5. Instructional Delivery, pp 40-44, Division of Undergraduate Science, Engineering, and Mathematics Education, Directorate for Science and Engineering Education, National Science Foundation, April 1989.

Gibbs, N. and A. Tucker, "A Model Curriculum for a Liberal Arts Degree in Computer Science", Communications of the ACM 29, 3 (1986), 202-210.

Gagne, R. and Briggs, L. Principles of Instructional Design, Holt, Rinehart and Winston, New York NY, 1979.

Halbert, D. and O'Brien, P., "Using Types and Inheritance in Object-Oriented Programming", IEEE Software, September 1987, pp. 71-79.

Jacobson, I., "Industrial Development of Software with an Object-Oriented Technique", Journal of Object-Oriented Programming, v. 4, No. 1, pp. 30-40, March/April 1991.

Khoshafian, S. and Copeland, G., "Object Identity", OOPSLA Conference Proceedings, New Orleans, Louisiana, October 1-6, 1986, pp. 406-415.

Knudsen, J., "Name Collision in Multiple Classification Hierarchies", ECOOP '88, European Conference on Object-Oriented Programming Proceedings, Oslo Norway, August 1988, pp. 93-109.

Lehman, M. Programs, "Life Cycles and Laws of Software Evolution", Proceedings of the IEEE No. 68, v(9), 1980.

Lientz, Swanson, and Tompkins, "Characteristics of Application Software Maintenance", Communication of the ACM, No. 21, v (6), 1978.

Loomis, M., "Integrating Objects with Relational Technology", Object Magazine, v. 1, No. 2, July/August 1991, pp 46-60.

Lutz, M. "Experiences With an Undergraduate Seminar on Object-Oriented Concepts", Proc SOOPPA 1990, pp 92-100, 1990.

Meyer, B., Object-Oriented Software Construction, Prentice Hall International, New York, NY, 1988.

Micallef, J., "Encapsulation, Reusability and Extendibility in Object-Oriented Programming Languages", Journal of Object-Oriented Programming, v. 1, No. 1, April/May 1988, pp. 12-35.

National Science Foundation, Executive Summary, "Courses and Curriculum", p. 5 Division of Undergraduate Science, Engineering, and Mathematics Education, Directorate for Science and Engineering Education, National Science Foundation, April 1989.

Reid, R. "Object-Oriented Programming in C++", SIGCSE Bulletin, Vol. 23, No. 2 June 1991.

Roberts, Eric, "Using C in CS1: Evaluating the Stanford Experience", Proceeding of the Twenty-Fourth SIGCSE Technical Symposium on Computer Science Education, March 1993.

Rumbaugh, J.; et. al., Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Saunders, John, "A Survey of Object-Oriented Programming Languages", Journal of Object-Oriented Programming, March/April, 1989.

Schonberg, E., "Contrasts: Ada 9x and C++", CrossTalk, September 1992.

Shackelford, R. and LeBlanc, R. "Integrating 'Depth First' and 'Breadth First' Models of Computing Curricula", SIGCSE Bulletin, Vol 26, No. 1, March 1994.

Shaw, M. "The Carnegie Mellon Curriculum for Undergraduate Computer Science", Springer Verlag New York, 1984.

Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA Conference Proceedings, Portland, Oregon, September 29 - October 2, 1986.

Stifik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations", The AI Magazine, Winter 1986, v. 6, No. 4, pp. 40-62.

Stroustrup, B, The C++ Programming Language, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Murray Hill, New Jersey, 1991.

Temte, M. "Let's Begin Introducing the Object-Oriented Paradigm", SIGCSE, Vol. 23, No. 1, pp 73-78, 1991.

Tucker, A. Wegner, P. "New Directions in the Introductory Computer Science Curriculum", SIGCSE Bulletin, Vol 26, No. 1. March, 1994.

Wasserman, A., "Object-Oriented Software Development: Issues in Reuse", Journal of Object-Oriented Programming, v. 4, No. 2, May 1991, pp. 55-57.

Wegner, P. and Zdonik, S., "Inheritance as an Incremental Modification or What like is and Isn't Like", ECOOP '88, European Conference on Object-Oriented Programming Proceeding, Oslo, Norway, August 1988, pp 55-77.

Wu, C. Thomas, "Teaching OOP to Beginners", Journal of Object-Oriented Programming, v6 p47(4) March-April 1993.

INITIAL DISTRIBUTION LIST

	Number of Copies
1. Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2. Library, Code 052 Naval Postgraduate School Monterey, California 93943-5002	2
3. Computer Technology, Code 32 Naval Postgraduate School Monterey, California 93943-5002	1
4. C. Thomas Wu, Code 32 Department of Computer Sciences Naval Postgraduate School Monterey, California 93943-5002	1
5. Lt. Curtis H. Loehr 742 Shady Oaks Dr. Coldwater Michigan, 49036	1